Python supports integers, floating point numbers and complex numbers. They are defined as int, float and complex class in Python.

Integers and floating points are separated by the presence or absence of a decimal point. 5 is integer whereas 5.0 is a floating point number.

Complex numbers are written in the form, x + yj, where x is the real part and y is the imaginary part.

We can use the type() function to know which class a variable or a value belongs to and isinstance() function to check if it belongs to a particular class.

a = 5

# Output: <class 'int'>

print(type(a))

# Output: <class 'float'>

print(type(5.0))

# Output: (8+3j)

c = 5 + 3j

print(c + 3)

# Output: True

print(isinstance(c, complex))

When you run the program, the output will be:

107

253

13

**Type Conversion**

We can convert one type of number into another. This is also known as coercion.

Operations like addition, subtraction coerce integer to float implicitly (automatically), if one of the operand is float.

>>> 1 + 2.0

3.0

We can see above that 1 (integer) is coerced into 1.0 (float) for addition and the result is also a floating point number.

We can also use built-in functions like int(), float() and complex() to convert between types explicitly. These function can even convert from [strings](https://www.programiz.com/python-programming/string).

>>> int(2.3)

2

>>> int(-2.8)

-2

>>> float(5)

5.0

>>> complex('3+5j')

(3+5j)

When converting from float to integer, the number gets truncated (integer that is closer to zero).

**Python Decimal**

Python built-in class float performs some calculations that might amaze us. We all know that the sum of 1.1 and 2.2 is 3.3, but Python seems to disagree.

>>> (1.1 + 2.2) == 3.3

False

What is going on?

It turns out that floating-point numbers are implemented in computer hardware as binary fractions, as computer only understands binary (0 and 1). Due to this reason, most of the

decimal fractions we know, cannot be accurately stored in our computer.

Let's take an example. We cannot represent the fraction 1/3 as a decimal number.

This will give 0.33333333... which is infinitely long, and we can only approximate it.

Turns out decimal fraction 0.1 will result into an infinitely long binary fraction of 0.000110011001100110011... and our computer only stores a finite number of it.

This will only approximate 0.1 but never be equal. Hence, it is the limitation of our computer hardware and not an error in Python.

>>> 1.1 + 2.2

3.3000000000000003

To overcome this issue, we can use decimal module that comes with Python. While floating point numbers have precision up to 15 decimal places, the decimal module has user settable precision.

import decimal

# Output: 0.1

print(0.1)

# Output: Decimal('0.1000000000000000055511151231257827021181583404541015625')

print(decimal.Decimal(0.1))

This module is used when we want to carry out decimal calculations like we learned in school.

It also preserves significance. We know 25.50 kg is more accurate than 25.5 kg as it has two significant decimal places compared to one.

from decimal import Decimal as D

# Output: Decimal('3.3')

print(D('1.1') + D('2.2'))

# Output: Decimal('3.000')

print(D('1.2') \* D('2.50'))

Notice the trailing zeroes in the above example.

We might ask, why not implement Decimal every time, instead of float? The main reason is efficiency. Floating point operations are carried out must faster than Decimal operations.

**When to use Decimal instead of float?**

We generally use Decimal in the following cases.

* When we are making financial applications that need exact decimal representation.
* When we want to control the level of precision required.
* When we want to implement the notion of significant decimal places.
* When we want the operations to be carried out like we did at school

**Python Fractions**

Python provides operations involving fractional numbers through its fractions module.

A fraction has a numerator and a denominator, both of which are integers. This module has support for rational number arithmetic.

We can create Fraction objects in various ways.

import fractions

# Output: 3/2

print(fractions.Fraction(1.5))

# Output: 5

print(fractions.Fraction(5))

# Output: 1/3

print(fractions.Fraction(1,3))

While creating Fraction from float, we might get some unusual results. This is due to the imperfect binary floating point number representation as discussed in the previous section.

Fortunately, Fraction allows us to instantiate with string as well. This is the preferred options when using decimal numbers.

import fractions

# As float

# Output: 2476979795053773/2251799813685248

print(fractions.Fraction(1.1))

# As string

# Output: 11/10

print(fractions.Fraction('1.1'))

This datatype supports all basic operations. Here are few examples.

from fractions import Fraction as F

# Output: 2/3

print(F(1,3) + F(1,3))

# Output: 6/5

print(1 / F(5,6))

# Output: False

print(F(-3,10) > 0)

# Output: True

print(F(-3,10) < 0)

**Python Mathematics**

Python offers modules like math and random to carry out different mathematics like trigonometry, logarithms, probability and statistics, etc.

import math

# Output: 3.141592653589793

print(math.pi)

# Output: -1.0

print(math.cos(math.pi))

# Output: 22026.465794806718

print(math.exp(10))

# Output: 3.0

print(math.log10(1000))

# Output: 1.1752011936438014

print(math.sinh(1))

# Output: 720

print(math.factorial(6))

import random

# Output: 16

print(random.randrange(10,20))

x = ['a', 'b', 'c', 'd', 'e']

# Get random choice

print(random.choice(x))

# Shuffle x

random.shuffle(x)

# Print the shuffled x

print(x)

# Print random element

print(random.random())

import math

math.sqrt(4)

|  |  |
| --- | --- |
| Function | Description |
| ceil(x) | Returns the smallest integer greater than or equal to x. |
| copysign(x, y) | Returns x with the sign of y |
| fabs(x) | Returns the absolute value of x |
| factorial(x) | Returns the factorial of x |
| floor(x) | Returns the largest integer less than or equal to x |
| fmod(x, y) | Returns the remainder when x is divided by y |
| frexp(x) | Returns the mantissa and exponent of x as the pair (m, e) |
| fsum(iterable) | Returns an accurate floating point sum of values in the iterable |
| isfinite(x) | Returns True if x is neither an infinity nor a NaN (Not a Number) |
| isinf(x) | Returns True if x is a positive or negative infinity |
| isnan(x) | Returns True if x is a NaN |
| ldexp(x, i) | Returns x \* (2\*\*i) |
| modf(x) | Returns the fractional and integer parts of x |
| trunc(x) | Returns the truncated integer value of x |
| exp(x) | Returns e\*\*x |
| expm1(x) | Returns e\*\*x - 1 |
| log(x[, base]) | Returns the logarithm of x to the base (defaults to e) |
| log1p(x) | Returns the natural logarithm of 1+x |
| log2(x) | Returns the base-2 logarithm of x |
| log10(x) | Returns the base-10 logarithm of x |
| pow(x, y) | Returns x raised to the power y |
| sqrt(x) | Returns the square root of x |
| acos(x) | Returns the arc cosine of x |
| asin(x) | Returns the arc sine of x |
| atan(x) | Returns the arc tangent of x |
| atan2(y, x) | Returns atan(y / x) |
| cos(x) | Returns the cosine of x |
| hypot(x, y) | Returns the Euclidean norm, sqrt(x\*x + y\*y) |
| sin(x) | Returns the sine of x |
| tan(x) | Returns the tangent of x |
| degrees(x) | Converts angle x from radians to degrees |
| radians(x) | Converts angle x from degrees to radians |
| acosh(x) | Returns the inverse hyperbolic cosine of x |
| asinh(x) | Returns the inverse hyperbolic sine of x |
| atanh(x) | Returns the inverse hyperbolic tangent of x |
| cosh(x) | Returns the hyperbolic cosine of x |
| sinh(x) | Returns the hyperbolic cosine of x |
| tanh(x) | Returns the hyperbolic tangent of x |
| erf(x) | Returns the error function at x |
| erfc(x) | Returns the complementary error function at x |
| gamma(x) | Returns the Gamma function at x |
| lgamma(x) | Returns the natural logarithm of the absolute value of the Gamma function at x |
| pi | Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...) |
| e | mathematical constant e (2.71828...) |